Draft

Clarification of UsdLux Quantities and Behavior, with Reference Implementation

Overview

The current specifications of the various UsdLux prims + attributes are imprecise or vague in many places, and as a result, actual implementations of them by various renderers have diverged, sometimes quite significantly.

For instance, here is Intel's 4004 Moore Lane scene, with the same UsdLux lights defined, in 3 different renderers:

Karma	Arnold	Omniverse RTX

To address this, we propose making two changes, in two stages: an update to documentation, and the inclusion of a reference renderer implementation in the main OpenUSD repository.

Stage 1: Documentation Clarification

We propose that the documentation for the UsdLux schema should be updated, such that the behavior of all UsdLux attributes are defined in precise terms. The goal would be to eliminate ambiguity and ensure that all renderers which conform to these definitions will produce similar results.

Backward Compatibility Notes

Support for UsdLux-defined lights has already been added to several renderers. Unfortunately, due to the lack of precise definitions in the current schema, each renderer has had to make their own implementation decisions, resulting in divergent behavior. Therefore, any existing UsdLux assets have an implicit dependency on the renderer they were "designed" for.

We expect/hope renderers will update their implementations to conform to the new, more precise definitions in this proposal by default. Since the exact details on current UsdLux implementation are renderer specific, we leave it up to each renderer to decide how best to deal with compatibility with existing UsdLux assets designed with the "old" ambiguous behavior in mind. However, if they opt to support some form of backward compatibility for existing assets, we can provide some guidelines:

- Renderers can provide a renderer-specific configuration option to specify whether UsdLux should use "old" (ambiguously-defined) behavior or "new" (precisely defined, in accordance with this proposal) behavior
- By default, renderers should use "new" behavior
- For instance, a hypothetical HappyRenderer might:
  - First check if the rendered stage has a RenderSettings node with a "happy:usdlux_old_compatibility" boolean attribute set, and if authored, always use "old" behavior if true, and "new" behavior if false for that stage.
  - If no RenderSetting attribute is set, it could check whether a "HAPPY_USDLUX_OLD_COMPATIBILITY" environment var is set, and if so, use that to determine default behavior
  - If no environment var is set, it could check for the value of a "usdlux_old_compatibility" configuration option in it's renderer configuration files, located at `~/.happy/config.toml` and `/etc/happy/config.yaml`.
  - Otherwise, assume "new" behavior

As mentioned above, we expect most existing usage of UsdLux stages to be "site internal" to a specific company or project. By providing a way to control default behavior and have explicit overrides, it allows such entities several options for how to handle compatibility / transition:

- They could opt to do a one-time mass conversion of all existing rendered stages to be explicitly tagged as having "usdlux_old_compatibility"
- They could opt to alter configuration on a per-project basis, if they have project-wide renderer configuration files
- They could specify "old" behavior by default in a site-wide renderer configuration file. Going forward, all new rendered stages could be explicitly authored as using the "new

behavior". At some point in the future when no "old" stages are in use, they could switch the site-wide default to "new", and stop explicitly authoring compatibilty information on new stages.

Updated Schema Classes and Attributes

Here are the suggested updates to the documentation in `usdLux/schema.usda` :

LightAPI

- **Non-Attribute Specific: Units**

  Previously, the UsdLux / LightAPI was intentionally unitless. While this in theory made UsdLux "broadly applicable" to many different renderers - which may have different default units or be unitless themselves - it creates problems with general interchange, particularly between renderers. Also, it creates ambiguity when mixing unitless lights with light sources that are defined in physical units (for instance, ies files).

  Therefore, we propose introducing SI photometric units in the the UsdLux specification - lumen, candela, and lux. (We will also sometimes refer to luminance measured in "nits" - though not a formal SI unit, a nit is defined to be 1 candela / meter^2).

  More precisely, we propose introducing this text into the documentation for `LightAPI` :

  > **Quantities and Units**
  >
  > Most renderers consuming OpenUSD today are RGB renderers, rather than spectral. Units in RGB renderers are tricky to define as each of the red, green and blue channels transported by the renderer represents the convolution of a spectral exposure distribution, e.g. CIE Illuminant D65, with a sensor response function, e.g. CIE 1931 $\bar{x}$. Thus the main quantity in an RGB renderer is neither radiance nor luminance, but "integrated radiance" or "tristimulus weight".
  >
  > The emission of a default light with `intensity` 1 and `color` [1, 1, 1] is an Illuminant D spectral distribution with chromaticity matching the rendering color space white point, normalized such that a ray normally incident upon the sensor with EV0 exposure settings will generate a pixel value of [1, 1, 1] in the rendering color space.

Given the above definition, that means that the luminance of said default light will be 1 *nit (cd/m²)* and its emission spectral radiance distribution is easily computed by appropriate normalization.

For brevity, the term *emission* will be used in the documentation to mean "emitted spectral radiance" or "emitted integrated radiance/tristimulus weight", as appropriate.

The method of "uplifting" an RGB color to a spectral distribution is unspecified other than that it should round-trip under the rendering illuminant to the limits of numerical accuracy.

Note that some color spaces, most notably ACES, define their white points by chromaticity coordinates that do not exactly line up to any value of a standard illuminant. Because we do not define the method of uplift beyond the round-tripping requirement, we discourage the use of such color spaces as the rendering color space, and instead encourage the use of color spaces whose white point has a well-defined spectral representation, such as D65.

- Attribute: `inputs:intensity`

  Scales the brightness of the light linearly.

  Expresses the "base", unmultiplied luminance emitted (L) of the light, in nits (cd/m²):

  $$L_{Scalar} = \text{intensity}$$

  Normatively, the lights' emission is in units of spectral radiance normalized such that a directly visible light with `intensity` 1 and `exposure` 0 normally incident upon the sensor plane will generate a pixel value of [1, 1, 1] in an RGB renderer, and thus have a luminance of 1 nit. A light with `intensity` 2 and `exposure` 0 would therefore have a luminance of 2 nits.

- Attribute: `inputs:exposure`

  Scales the brightness of the light exponentially as a power of 2 (similar to an F-stop control over exposure). The result is multiplied against the intensity:

  $$L_{Scalar} = L_{Scalar} \cdot 2^{exposure}$$

Normatively, the lights' emission is in units of spectral radiance normalized such that a directly visible light with `intensity` 1 and `exposure` 0 normally incident upon the sensor plane will generate a pixel value of [1, 1, 1] in an RGB renderer, and thus have a luminance of 1 nit (cd/m²). A light with `intensity` 1 and `exposure` 2 would therefore have a luminance of 4 nits.

- Attribute: `inputs:normalize`

  Normalizes the emission such that the power of the light remains constant while altering the size of the light, by dividing the luminance by the world-space surface area of the light.

  This makes it easier to independently adjust the brightness and size of the light, by causing the total illumination provided by a light to not vary with the area or angular size of the light.

  Mathematically, this means that the luminance of the light will be divided by a factor representing the "size" of the light:

  $$L_{Scalar} = L_{Scalar} / sizeFactor$$

  ...where `sizeFactor` = 1 if `normalize` is off, and is calculated depending on the family of the light as described below if `normalize` is on.

  ## DomeLight / PortalLight:

  For a dome light (and its associated PortalLight), this attribute is ignored:

  $$sizeFactor_{dome} = 1$$

  ## Area Lights:

  For an area light, the `sizeFactor` is the surface area (in world space) of the shape of the light, including any scaling applied to the light by its transform stack. This includes the boundable light types which have a calculable surface area:

  - MeshLightAPI
  - DiskLight
  - RectLight
  - SphereLight

- CylinderLight

$$\text{sizeFactor}_{\text{area}} = \text{worldSpaceSurfaceArea(light)}$$

DistantLight:

For distant lights, we first define θ_{max} as:

$$\theta_{\text{max}} = \text{clamp(toRadians(distantLightAngle) / 2, 0, }\pi)$$

Then we use the following formula:

- *if θ_{max} = 0:*

$$\text{sizeFactor}_{\text{distant}} = 1$$

- *if $0 < \theta_{max} \leq \pi / 2$:*

$$\text{sizeFactor}_{\text{distant}} = \sin^2\theta_{\text{max}} \cdot \pi$$

- *if $\pi / 2 < \theta_{max} \leq \pi$:*

$$\text{sizeFactor}_{\text{distant}} = (2 - \sin^2\theta_{\text{max}}) \cdot \pi$$

This formula is used because it satisfies the following two properties:

i. When normalize is enabled, the received illuminance from this light on a surface normal to the light's primary direction is held constant when angle changes, and the "intensity" property becomes a measure of the illuminance, expressed in lux, for a light with 0 exposure.

ii. If we assume that our distant light is an approximation for a "very far" sphere light (like the sun), then (for *$0 < \theta_{max} \leq \pi/2$*) this definition agrees with the definition used for area lights - i.e., the total power of this distant sphere light is constant when the "size" (i.e., angle) changes, and our sizeFactor is proportional to the total surface area of this sphere.

Other Lights

The above taxonomy describes behavior for all built-in light types. (Note that the above is based on schema *family* - i.e., `DomeLight_1` follows the rules for a `DomeLight`, and ignores `normalize`).

Lights from other third-party plugins / schemas must document their expected behavior with regards to normalize. However, some general guidelines are:

- Lights that either inherit from or are strongly associated with one of the built-in types should follow the behavior of the built-in type they inherit/resemble; i.e., a renderer-specific "MyRendererRectLight" should have its size factor be its world-space surface area.
  - Lights that are boundable and have a calculable surface area should follow the rules for an Area Light, and have their sizeFactor be their world-space surface area.
  - Lights that are non-boundable and/or have no way to concretely or even "intuitively" associate them with a "size" will ignore this attribute (and always set sizeFactor = 1).

  Lights that don't clearly meet any of the above criteria may either ignore the normalize attribute or try to implement support using whatever heuristic seems to make sense. For instance, MyMandelbulbLight might use a sizeFactor equal to the world-space surface area of a sphere which "roughly" bounds it.

- Attribute: `inputs:color`

  The color of emitted light, in the rendering color space.

  This color is just multiplied with the emission:

  $$L_{Color} = L_{Scalar} \cdot color$$

  In the case of a spectral renderer, this color should be uplifted such that it round-trips to within the limit of numerical accuracy under the rendering illuminant. We recommend the use of a rendering color space well defined in terms of a Illuminant D illuminant (ideally a D illuminant whose white point has a well-defined spectral representation, such as D65), to avoid unspecified uplift. See: \ref usdLux_quantities

- Attribute: `inputs:colorTemperature`

  Color temperature, in degrees Kelvin, representing the white point. The default is a common white point, D65. Lower values are warmer and higher values are cooler. The valid range is from 1000 to 10000. Only takes effect when enableColorTemperature is set to true. When active, the computed result multiplies against the color attribute. See UsdLuxBlackbodyTemperatureAsRgb().

This is always calculated as an RGB color using a D65 white point, regardless of the rendering color space, normalized such that the default value of 6500 will always result in white, and then should be transformed to the rendering color space.

Spectral renderers should do the same and then uplift the resulting color after multiplying with the `color` attribute. We recommend the use of a rendering color space well defined in terms of a Illuminant D illuminant, to avoid unspecified uplift. See: \ref usdLux_quantities

ShapingAPI

- Attribute: `inputs:shaping:focus`

  A control to shape the spread of light. Higher focus values pull light towards the center and narrow the spread.

  This is implemented as a multiplication with the absolute value of the dot product between the light's surface normal and the emission direction, raised to the power `focus`. See `inputs:shaping:focusTint` for the complete formula, but if we assume a default `focusTint` of pure black, then that formula simplifies to:

  $$\text{focusFactor} = |\text{ emissionDirection} \cdot \text{lightNormal} |^{\text{focus}}$$
  $$L_{\text{Color}} = \text{focusFactor} \cdot L_{\text{Color}}$$

  Values < 0 are ignored.

- Attribute: `inputs:shaping:focusTint`

  Off-axis color tint. This tints the emission in the falloff region. The default tint is black.

  This is implemented as a linear interpolation between `focusTint` and white, by the factor computed from the focus attribute, in other words:

  $$\text{focusFactor} = |\text{ emissionDirection} \cdot \text{lightNormal} |^{\text{focus}}$$
  $$\text{focusColor} = \text{lerp(focusFactor, focusTint, [1, 1, 1])}$$

  $$L_{\text{Color}} = \text{componentwiseMultiply(focusColor, } L_{\text{Color}})$$

  Note that this implies that a focusTint of pure white will disable focus.

- Attribute: `inputs:shaping:cone:angle`

Angular limit off the primary axis to restrict the light spread, in degrees.

Light emissions at angles off the primary axis greater than this are guaranteed to be zero, i.e.:

$$\theta_{offAxis} = acos(lightAxis \bullet emissionDir)$$
$$\theta_{cutoff} = toRadians(coneAngle)$$

$$\mathbf{\theta_{offAxis} > \theta_{cutoff}}$$
$$\mathbf{\implies L_{Scalar} = 0}$$

For angles < coneAngle, see the documentation for `shaping:cone:softness`. However, at the default of coneSoftness = 0, the luminance is unaltered if emissionOffAxisAngle <= coneAngle, so the coneAngle functions as a hard binary "off" toggle for all angles > coneAngle.

- Attribute: `inputs:shaping:cone:softness`

  Controls the cutoff softness for cone angle.

  At the default of coneSoftness = 0, the luminance is unaltered if emissionOffAxisAngle <= coneAngle, and 0 if emissionOffAxisAngle > coneAngle, so in this situation the coneAngle functions as a hard binary "off" toggle for all angles > coneAngle.

  For coneSoftness in the range (0, 1], it defines the proportion of the non-cutoff angles over which the luminance is smoothly interpolated from 0 to 1. Mathematically:

  $$\theta_{offAxis} = acos(lightAxis \bullet emissionDir)$$
  $$\theta_{cutoff} = toRadians(coneAngle)$$

  $$\theta_{smoothStart} = lerp(coneSoftness, \theta_{cutoff}, 0)$$

  $$L_{Scalar} = L_{Scalar} \cdot (1 - smoothStep(\theta_{offAxis}, \theta_{smoothStart}, \theta_{cutoff}))$$

  Values outside of the [0, 1] range are clamped to the range.

- Attribute: `inputs:shaping:ies:file`

An IES (Illumination Engineering Society) light profile describing the angular distribution of light.

For full details on the .ies file format, see the full specification, ANSI/IES LM-63-19:

https://store.ies.org/product/lm-63-19-approved-method-ies-standard-file-format-for-the-electronic-transfer-of-photometric-data-and-related-information/

The luminous intensity values in the IES profile are sampled using the emission direction in the light's local space (after a possible transformation by a non-zero shaping:ies:angleScale, see below). The sampled value is then potentially normalized by the overall power of the profile if shaping:ies:normalize is enabled, and then used as a scaling factor on the returned luminance:

$$\theta_{light}, \phi = \text{toPolarCoordinates(emissionDirectionInLightSpace)}$$
$$\theta_{ies} = \text{applyAngleScale}(\theta_{light}, \text{angleScale})$$

$$\text{iesSample} = \text{sampleIES}(\text{iesFile}, \theta_{ies}, \phi)$$

$$\text{iesNormalize} \implies \text{iesSample} = \text{iesSample} \cdot \text{iesProfilePower(iesFile)}$$

$$L_{Color} = \text{iesSample} \cdot L_{Color}$$

See `inputs:shaping:ies:angleScale` for a description of `applyAngleScale`, and `inputs:shaping:ies:normalize` for how `iesProfilePower` is calculated.

- Attribute: `inputs:shaping:ies:angleScale`

  Rescales the angular distribution of the IES profile.

  Applies a scaling factor to the latitudinal theta/vertical polar coordinate before sampling the IES profile, to shift the samples more toward the "top" or "bottom" of the profile. The scaling origin varies depending on whether `angleScale` is positive or negative. If it is positive, the scaling origin is theta = 0. If it is negative, the scaling origin is theta = pi (180 degrees). Values where |angleScale| < 1 will "shrink" the angular range in which the iesProfile is applied, while values where |angleScale| > 1 will "grow" the angular range to which the iesProfile is mapped.

  If θ_{light} is the latitudinal theta polar coordinate of the emission direction in the light's local space, and θ_{ies} is the value that will be used when actually sampling the profile, then the exact formula is:

- *if angleScale > 0:*

$$\theta_{ies} = \theta_{light} \, / \, \mathbf{angleScale}$$

- *if angleScale = 0:*

$$\theta_{ies} = \theta_{light}$$

- *if angleScale < 0:*

$$\theta_{ies} = (\theta_{light} - \pi) \, / \, \mathbf{\text{-angleScale}} + \pi$$

Usage guidelines for artists / lighting TDs:

If you have an IES profile for a spotlight aimed "down":

- You should use a positive angleScale (> 0).
- Values where 0 < angleScale < 1 will narrow the spotlight beam.
- Values where angleScale > 1 will broaden the spotlight beam.

For example, if the original IES profile is a downward spotlight with a total cone angle of 60°, then angleScale = .5 will narrow it to have a cone angle of 30°, and an angleScale of 1.5 will broaden it to have a cone angle of 90°.

If you have an IES profile for a spotlight aimed "up":

- You should use a negative angleScale (< 0).
- Values where -1 < angleScale < 0 will narrow the spotlight beam.
- Values where angleScale < -1 will broaden the spotlight beam.

For example, if the original IES profile is an upward spotlight with a total cone angle of 60°, then angleScale = -.5 will narrow it to have a cone angle of 30°, and an angleScale of -1.5 will broaden it to have a cone angle of 90°.

If you have an IES profile that's isn't clearly "aimed" in a single direction, OR it's aimed in a direction other than straight up or down:

- Applying angleScale will alter the vertical angle mapping for your IES light, but it may be difficult to have a clear intuitive sense of how varying the angleScale will affect the shape of your light

If you violate the above rules (i.e., use a negative angleScale for a spotlight aimed down), then angleScale will still alter the vertical- angle mapping, but in more non-intuitive ways (i.e., broadening / narrowing may seem inverted, and the IES profile may seem to "translate" through the vertical angles, rather than uniformly scale).

- Attribute: `inputs:shaping:ies:normalize`

   > Normalizes the IES profile so that it affects the shaping of the light while preserving the overall energy output.
   >
   > The sampled luminous intensity is scaled by the overall power of the IES profile if this is on, where the total power is calculated by integrating the luminous intensity over all solid angle patches defined in the profile.

DistantLight

- Attribute: `inputs:angle`

   > Angular diameter of the light in degrees. As an example, the Sun is approximately 0.53 degrees as seen from Earth. Higher values broaden the light and therefore soften shadow edges.
   >
   > This value is assumed to be in the range `0 <= angle < 360`, and will be clipped to this range. Note that this implies that we can have a distant light emitting from more than a hemispherical area of light if angle > 180. While this is valid, it is possible that for large angles a DomeLight may provide better performance. If angle is 0, the DistantLight represents a perfectly parallel light source.

Alternative approaches

New Schemas (or new schema versions)

We could have opted to create an entirely new schema, possibly making use of USD's [schema versioning](). The advantage of this would be backward compatibility for existing UsdLux assets - any renderers which previously implemented any UsdLux behavior in a different way could retain their existing implementation for the "old" schema, and only provide new / unified behavior for a new schema.

However, we felt that it would be better to simply update the existing schema, for the following reasons:

- **Not Keeping Undefined Behavior**

In most cases, we are not changing defined behavior, but clarifying undefined or ambiguously defined behavior. If we leave the current schema "as is", then we are essentially committing to forever keeping the current schema's behavior undefined or poorly defined.

- **C++ Complexity of Supporting Multiple Schemas**

  If we introduce a completely new schema, it makes interacting with "generic" UsdLux objects difficult at a C++ level. For instance, if we add a new `UsdLuxCylinderLight_1`, all C++ methods that currently work with `UsdLuxCylinderLight` will have to be modified to work with `std::variant<UsdLuxCylinderLight, UsdLuxCylinderLight_1>`, or else they are both made to inherit from a `UsdLuxCylinderLightBase`, etc. In either case, it's a fairly large change to the existing APIs, both inside of the OpenUSD codebase, and for any 3rd-party render delegate implementations.

  Unfortunately, current USD [schema versioning](#) doesn't help with this - it only provides utilities for identifying if two schemas share the same "family", but no means for treating members of a family in a unified manner.

- **Relatively Low Current Adoption of UsdLux**

  Without having done any formal polling, our general sense is that adoption of UsdLux is still fairly low.

  The ambiguities in the current specification make UsdLux a poor fit for external interchange of lights, as there is no assurance that they will be interpreted in the same manner as originally intended.

  Thus we expect most existing usage of UsdLux assets to be "site internal" - ie, for private assets used within a company in it's own rendering pipeline. While it's hard to know the extent of such usage, such entities can employ tactics to ease transition that aren't applicable in the "general" case - see below for more details.

Reference Pull Request

- [PR3182](#): Accompanying OpenUSD Pull Request

Prior Work

- [PR2758](#): Old Pull Request (now superceded by [PR3182](#), above)

- [light_comparison](): Repository containing initial attempt to define the problem, and some test cases

Stage 2: Reference Implementation

Once we have provided clear guidance on expected behavior for UsdLux lights, we will update the included HdEmbree render delegate to include support for UsdLux lights. This will give a reference implementation to aid those wishing to provide UsdLux support in their renderers.

Performance vs Clarity

The goal of the UsdLux support in HdEmbree will be as a reference, not for it to be used as a production renderer; therefore, it will not be performance optimized, and in fact will prefer clarity and simplicity in it's code over performance.

Supported Light Types

- DistantLight
- DiskLight
- RectLight
- SphereLight
- CylinderLight
- DomeLight

Unsupported Lights and Features

- MeshLightAPI
- GeometryLight
- PortalLight
- PluginLight

Reference Pull Requests

To aid with review and adoption, the work to add UsdLux support to the HdEmbree render delegate was broken into a chain of smaller PRs:

- Collapsed:

- - [PR3199](): Combination of all Embree UsdLux Reference Implementation PRs
- Separate PRs:
  - [PR3211](): [hdEmbree] fix for random number generation (hdEmbree-UsdLux-PR01)
  - [PR3183](): [hdEmbree] add HDEMBREE_RANDOM_NUMBER_SEED (hdEmbree-UsdLux-PR02)
  - [PR3185](): [hdEmbree] minor fixes / tweaks (hdEmbree-UsdLux-PR03)
  - [PR3234](): [hdEmbree][build_usd] add to build_usd.py status message (hdEmbree-UsdLux-PR04)
  - [PR3198](): [hdEmbree] ensure we respect PXR_WORK_THREAD_LIMIT (hdEmbree-UsdLux-PR05)
  - [PR3186](): [hdEmbree] Initial UsdLux reference implementation (hdEmbree-UsdLux-PR06)
  - [PR3196](): [hdEmbree] add HDEMBREE_LIGHT_CREATE debug code (hdEmbree-UsdLux-PR07)
  - [PR3195](): [hdEmbree] add support for lighting double-sided meshes (hdEmbree-UsdLux-PR08)
  - [PR3197](): [hdEmbree] add support for inputs:diffuse (hdEmbree-UsdLux-PR09)
  - [PR3187](): [hdEmbree] add light texture support (hdEmbree-UsdLux-PR10)
  - [PR3188](): [hdEmbree] add dome light suppport (hdEmbree-UsdLux-PR11)
  - [PR3189](): [hdEmbree] add direct camera visibility support for rect lights (hdEmbree-UsdLux-PR12)
  - [PR3190](): [hdEmbree] add pxrPbrt/pbrUtils.h (hdEmbree-UsdLux-PR13)
  - [PR3191](): [hdEmbree] add distant light support (hdEmbree-UsdLux-PR14)
  - [PR3192](): [hdEmbree] Add utilities for processing ies light files (hdEmbree-UsdLux-PR15)
  - [PR3193](): [hdEmbree] Add a PxrIESFile class which mimics iesFile, with additional functionality (hdEmbree-UsdLux-PR16)
  - [PR3194](): [hdEmbree] add lighting support for IES files (hdEmbree-UsdLux-PR17)

Related Work

- [luxtest](): Various UsdLux test scenes and aids for rendering + comparison of results